Not only computing – also art

JOHN LANSDOWN

Ignorance, Madam, pure ignorance

Beverley Rowe, one of the founder members of the Computer Arts Society, has forcibly – but kindly – pointed out to me that I was talking nonsense about probability in the December 1984 issue of *Computer Bulletin*. He's right and I didn't write what I meant. I suppose I was trying to say no more than 'Just as some probabilities need to be added and some need to be multiplied, so some fuzzy variables need to be MAX-ed and some need to be MIN-ed'. What I actually said came out rather differently and seemed to imply that you could go on adding probabilities indefinitely, which is absurd. I can only give the excuse that Dr Johnson gave to the lady who asked him why, in his dictionary, he'd wrongly defined something as 'the knee of a horse'.

Patterns

Pattern-making is not so widespread an art form in the West as it is in the countries of the East and the Middle East. Indeed, over here, the process is often frowned upon and characterised as 'mere pattern-making'. To my mind, this view is mistaken. I believe that the generation of patterns is a potentially rich form of art which should be encouraged. To produce aesthetically pleasing patterns requires creative thinking. The results often exploit and illuminate in a very pure way the Gestalt principles of pattern recognition and visual organisation which we seem to employ when looking at pictures. Those concerned with teaching art to school-children, in particular, should look on programs for pattern recognition as an important resource. Ideas of symmetry, similarity, and continuity as well as the relationships between patterns and mathematics can be readily and engagingly taught using such programs.

In addition, using computers for pattern generation need not involve expensive hardware. Reader, Edward Collier from Ludlow, has sent me examples of some of his output from a program he has written for his Sinclair Spectrum driving an Epsom RX80F/T+ printer. The program, called Patchwork, is interactive and, as Edward says, 'Once a pattern has been produced on the screen in a few seconds, it can be manipulated to make it symmetrical or a window can be moved to encompass the most interesting section of the pattern and this in turn can be magnified to fill the screen. An alternative way of producing patterns is to magnify characters (keyboard or user-defined) and then to make the resultant screen display symmetrical'. The effects of these manipulations are striking and two are shown as Figures 1 and 2.

Taking an eyeful

At the other end of the scale, from the points of view both of picture-making and computer systems, are the drawings shown in Figure 3. These are by Keith Waters of Middlesex Polytechnic's Faculty of Art and Design and won top prize in the Student Division of Calcomp's 25th Anniversary International Computer Art Competition. The drawings, titled 'Effie', are a selection from a sequence of views of the Eiffel Tower and were created using John Vince's Picaso software. (Incidentally, on the strength of this and his other work such as that in Figure 4, Keith has just been offered a place on the post-graduate course at that hot-bed of high quality computer

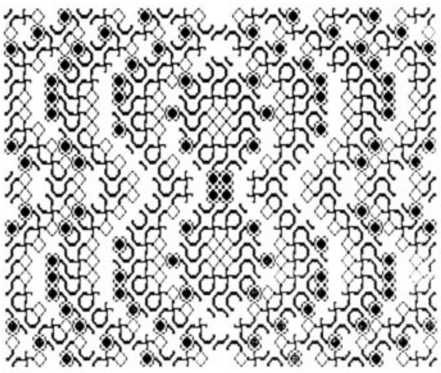
Figure 1

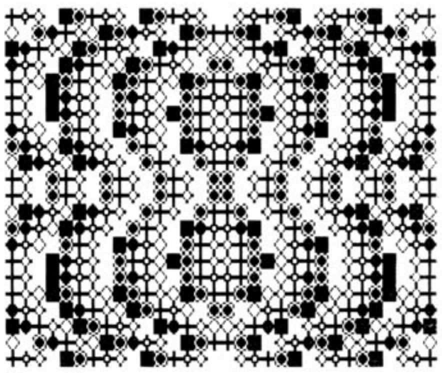
Figure 2

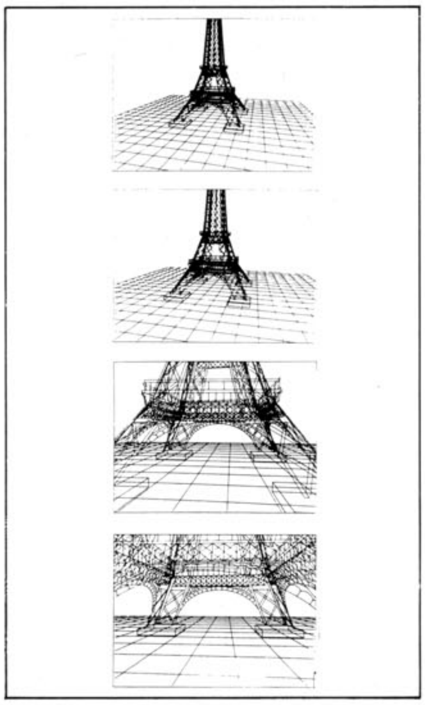
Figure 3

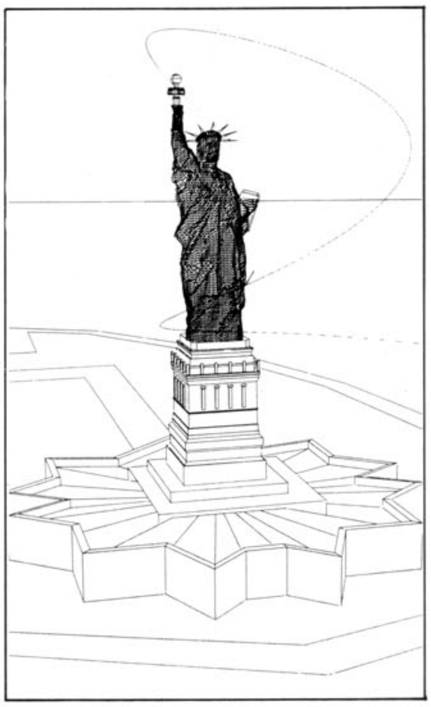
Figure 4

graphics, the Cranston Centre, Ohio State University.)

What is interesting to me about these drawings and those of Edward Collier is that they tend towards the same degree of complexity as realistic images without themselves being realistic. Of course, Keith's drawings are more like an identifiable object than Edward's. But, for example, the actual Eiffel Tower is made from solid metal and is not transparent – as Keith's drawings might lead a Martian to believe. Both the 'realistic' and the 'non-realistic' drawings are, in fact, conventions – something we often tend to forget.

There is a debate going on in computer graphics circles on just how much realism we should aim for. (This is not strictly true. Let me say instead, I have been trying for some time to *start* a debate in computer graphics circles on just how much realism we should aim for.) After some years of thinking otherwise, I no longer believe that we can achieve general photographic realism by the techniques we currently employ. Some objects – particularly man-made objects – can be rendered in ways which can fool us into thinking we are seeing photographs rather than computer drawings. In the general case, however, the complexity of textures, lighting, colours and forms that real scenes possess are beyond our capabilities (and are likely to remain so).

Already some drawings take inordinate amounts of time to process, even on supercomputers. It will be argued that a picture presently requiring two hours to create on a Cray I will need only 12 minutes, 1.2 minutes or 1.2 seconds on the next generation of machines. To this I reply, just compare a photograph of, say, a random street scene with a computer rendering of anything other than essentially 'unrealistic' objects – such as molecules or water splashes. The photograph exhibits a complexity which is orders of magnitude greater than that in the drawing. If my argument is correct, then we should, I think, be aiming at convincing naturalism rather than photographic realism. There are some cases where images need to be as realistic as they can be: for example, in simulations of lighting effect in room design and, perhaps, for special effects in films. But, for most other purposes, convincing naturalism, which exploits our remarkable perceptual abilities (for understanding such things as line drawings), is probably all we need. To paraphrase Einstein **Drawings should be as realistic as they need to be, but no more so.**

Correspondence *continued*

Comments from Alex M. Andrew

In my article I explored some possibilities arising from the fact that user-declared functions can be used recursively in the version of Basic implemented on the Sinclair Spectrum. I indicated that there are limitations due to the failure of the system to stack the parameters when function calls are nested, so that parameter values come to be overwritten. My attention was focussed on the type of recursion which is embodied in the function definition, as exemplified by recursive functions for computing factorials, highest common factors, and so on. There is, however, another type of recursion which appears only in the call of the function, and it is important to realise that there are limitations here also, attributable to the failure to stack parameter values. In this form they introduce a pitfall for the unwary which could be troublesome in everyday programming contexts.

Recursion appearing in the function call is often illustrated by reference to a simple non-recursive function to derive the maximum of two values. In the conventions of Algol68 this could take the form

```
PROC max=(REAL a, b) REAL:
BEGIN IF a > b THEN
a ELSE b FI END
```

A programmer wishing to obtain the maximum out of four values might then write

$x := \max(\max(a, b), \max(c, d))$

and in this he would be using PROC max recursively. The same thing can be done in Spectrum Basic, in which a maximum function can be realised either as on p131 of the manual

```
10 DEF FN m(x, y) =
   (x + y + ABS(x − y))/2
```

or using the stratagem I introduced in my article

```
5 DEF FN c$ (a$, x) = a$ (1 TO
  x* LEN a$)
10 DEF FN m(x,y) =
  VAL (FN c$("x", x > y)
  + FN c$( "y", x > = y))
```

To obtain the maximum out of four values it seems sensible to write

```
20 LET x = FN m(FN m(a, b), FN
  m(c, d))
```

This, however, will not have the desired effect. The outer call of the function invokes two inner calls to evaluate its parameters. The first of these evaluates the first parameter as the maximum value out of *a* and *b*. When the second inner call is made, this value is overwritten by the value of *c*. The overall result is that the value assigned to *x* is the maximum out of the two values *c* and *d*, which may or may not be the maximum out of the four values as intended. It is, of course, easy to find alternative formulations which avoid the error, but it is particularly pernicious in that the erroneous statement looks perfectly simple and natural, and also because the program will probably give correct results for some sets of input data. The golden rule for avoiding trouble is to avoid recursion in the evaluation of any except the first of multiple parameters. (Many versions of Basic only allow functions of single arguments, so the problem cannot arise. Two possible reformulations are as follows:

```
20 LET x = FN m(FN m(FN m(a,
  b), c), d)
```
and
```
20 LET x = FN m(a, b):
   LET x = FN m(x, c):
   LET x = FN m(x, d)
```

Techniques for computing the notorious Ackermann's Function in Basic have since been discussed by W. L. van der Poel and now H. J. Gawlik. In my article I was concerned with recursion achieved within single-line function definitions and these gentlemen have not achieved Ackermann's Function within this restriction. The functions I referred to as difficult (both Ackermann's and the Towers of Hanoi) are easily, though somewhat inelegantly, achieved if the programmer makes his own arrangements for stacking of variables. The final program presented by van der Poel shows a highly ingenious way of achieving a substantial depth of recursion without making much demand on storage, although his line 90 cannot ever have an effect. I have verified (on the Spectrum) that both of these ingenious programs operate correctly, and that the operation of Professor van der Poel's is unaffected when line 90 is deleted. That of Mr Gawlik requires slight modifications to run on a system where the array subscripts start from value one rather than zero.

The designers of the Spectrum are to be commended, I think, for implementing the VAL function in the way that makes my 'dirty trick' possible. It shows the right sort of bootstrapping approach to system design and it is a pity they did not go a little further by allowing the variables to be stacked.